A Few Very Useful Things:

- Comments and Documentation
- Functions and their Parameters
- Including Header Files with `#include`
- Standard Library Functions `printf()` and `scanf()`

User interface for the simple adder program

Basic building blocks of the C language:

- Keywords and identifiers
- Variables and constants
- Defining symbolic constants with `#define`

Basic data types in C: `int`, `char`, `double`
“hello world” Program Revisited

/* The traditional first program in honor of Dennis Ritchie who invented C at Bell Labs in 1972 */

#include <stdio.h>

int main()
{
    printf("hello world!\n");
    return 0;
}

In the following slides, we will review all these components!
Comments and Their Syntax

❖ A comment is any text found between /* ... */. Specifically, a comment starts with /* and ends with */.

❖ Comments do absolutely nothing, but are very useful for humans.

❖ Comments can span several lines (or several pages), but they cannot be nested:

```c
/* This is /* a very long and invalid (nested) */
 /* comment. */ /* A second comment. */
```

❖ Here is another valid way (introduced in C99) to include comments. Any text from // till the end of the same line:

```c
// This is a comment.
a = b + c; // This is another comment.
```
Functions in C

❖ In general, a function is “a convenient way to encapsulate some computation” which can then be invoked via a function call. Every function can do the following:
  ‣ Accept parameters
  ‣ Return a value

❖ Functions are called “procedures” or “routines” in other languages.

❖ Every function has a declaration and a definition as follows
  
  **Declaration:** describes the function’s name, the type of its parameters, and the type of value it returns.

  **Definition:** contains the declaration (again) as well as the code that actually implements the function’s computation.

Before a function can be invoked, its declaration must be given.

❖ We will talk much more about functions in this course!
Simple Function: An Example

**Declaration:**

```c
float power(float x, int n);
```

**Definition:**

```c
float power(float x, int n) {
    float p;
    int i;

    p = 1.0;
    for (i = 0; i < n; i++)
        p = p * x;
    return p;
}
```
Every C program must contain a function called `main`. Execution of the program begins with this function:

```c
int main()
{
    ...
}
```

We assume for the time being that `main()` has no parameters (in fact, it can have two parameters `argc` and `argv[]`).

**What is the value that `main()` returns and to whom?**

- The value is returned to the operating system, or to the process that called your entire program. It usually signifies a termination code.
- In case of normal (successful) termination, always `return 0`. Otherwise, return a nonzero value.
Standard *header files* such as `stdio.h`, `stdlib.h`, `math.h` contain *declarations* of many useful functions (e.g. `printf`) that Ritchie and Kernighan already implemented for you as part of the C *Standard Library*. If you include these files in the beginning of your program, you can invoke the corresponding functions.

You do not have to actually paste these files into your program. The `#include` compiler directive does the same thing:

```c
#include <stdio.h>  // declarations of input/output functions
#include <math.h>  // declarations of mathematical functions
```

Later on, you will learn how to write your own functions and make them available to all your programs using a header file.
Function `printf` from `stdio.h`

**Objective:** Print characters to the screen (more precisely to the `stdout` file stream).

**Parameters:** One *format string* and zero or more additional arguments. The format string can contain ordinary characters (which are just printed) and *conversion specifications*. Each conversion specification begins with a `%` and ends with a *conversion character*, such as: `d` (decimal `int`), `f` (floating-point real), `s` (string of characters), etc.

**Returned Value:** Total numbers of characters printed.

```c
printf("Just printing text. \n");
```

```c
int i = 5;
printf("The average of (%d,%d,%d) is %f", 1, i, 5, (1+i+5)/3.0);
```

**More Details:** [Kernighan & Ritchie], pp.153-155.
Objective: Read characters from the user screen (more precisely from the `stdin` file stream).

Parameters: One format string and zero or more additional arguments. The format string can contain white space (ignored), ordinary characters (must match the input exactly), and conversion specifications. Each conversion specification begins with `%` and ends with a conversion character, such as: `d` (decimal `int`), `f` (floating-point real value), `s` (string of characters), and more.

Important Note: Every argument of `scanf`, other than the format string, must be a pointer (address).

```c
int i; double x; float y;
scanf("%d%lf%f", &i, &x, &y);
```

Returned Value: Total number of values read (arguments converted) or –1 in case of failure before any successful conversions.
A few very useful things:
- Comments and documentation
- Functions and their parameters
- Including header files with `#include`
- Standard library functions `printf()` and `scanf()`

User Interface for the Simple Adder Program

Basic building blocks of the C language:
- Keywords and identifiers
- Variables and constants
- Defining symbolic constants with `#define`

Basic data types in C: `int`, `char`, `double`
Recall the Simple Adder Program

```c
#include <stdio.h>

int main()
{
    int i, sum, value;
    sum = 0;
    for (i = 0; i < 10; i++)
    {
        scanf("%d", &value);
        sum = sum + value;
    }
    printf("%d \n", sum);
    return 0;
}
```
```c
#include <stdio.h>

int main()
{
    int i, sum = 0, value, numbers_to_read;

    printf("Please enter number of values:\n");
    scanf("%d", &numbers_to_read);
    for (i = 0; i < numbers_to_read; i++)
    {
        printf("Enter the next integer: ");
        scanf("%d", &value);
        sum = sum + value;
    }
    printf("The sum of the %d numbers is %d.\n", numbers_to_read, sum);

    return 0;
}
```
Comments on the User Interface

❖ Note that the number of summands in the new “simple adder” program is determined by the user at run-time. Thus the new “simple adder” program is interactive.

❖ An interactive program is a program that conducts a dialogue with the user, via input and output designed for this purpose.

❖ An interactive program must be “user friendly.” It should explain to the user clearly what he/she is supposed to do and what is the exact meaning of the output the user is getting.

❖ What if the user enters a non-integer when prompted for the number of values to add?
   » Checking the validity of user’s input prevents serious problems at run-time, due to input errors. Don’t trust the user!
User Input Verification

```
...
printf("Please enter number of values:\n");
if (scanf("%d", &numbers_to_read) < 1)
{
    printf("Failed reading the number of values.\n");
    return 1;
}
for (i = 0; i < numbers_to_read; i++)
{
    printf("Enter the next integer: ");
    if (scanf("%d", &value) < 1)
    {
        printf("Failed reading value %d.\n", i + 1);
        return 1;
    }
    sum = sum + value;
}
...```
Outline of this Lecture

❖ A few very useful things:
  ‣ Comments and documentation
  ‣ Functions and their parameters
  ‣ Including header files with \#include
  ‣ Standard library functions printf() and scanf()

❖ User interface for the simple adder program

❖ **Basic Building Blocks of the C Language:**
  ‣ Keywords and Identifiers
  ‣ Variables and Constants
  ‣ Defining Symbolic Constants with \#define

❖ Basic data types in C: int, char, double
Every C program is simply a **sequence of characters**. The valid characters are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>lowercase letters</td>
<td>a b c d e f g h i j ... w x y z</td>
</tr>
<tr>
<td>uppercase letters</td>
<td>A B C D E F G H I J ... W X Y Z</td>
</tr>
<tr>
<td>digits</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>other characters</td>
<td>+ - * / = ( ) { } [ ] &lt; &gt; ` &quot;</td>
</tr>
<tr>
<td></td>
<td># ! % &amp; _</td>
</tr>
<tr>
<td>whitespace characters</td>
<td>blank, tab, newline, etc.</td>
</tr>
</tbody>
</table>

The first thing the C compiler does is: execute all the compiler directives (all lines beginning with # such as `#include`), then split the sequence of characters into **whitespace** and **tokens**.

There are six types of tokens: **keywords, identifiers, constants, string constants, operators, and punctuation**.
White Spaces in C

❖ A white space is a sequence consisting of one or more of the following:
   ‣ Comment (anything between /* and */ or after //)
   ‣ Whitespace character (blank, tab, newline, etc.)

❖ The compiler reduces each white space to a single blank. Thus white spaces are ignored, except for the purpose of serving as separators between tokens. For example:

\[
\text{sum=a+b; } \quad \text{sum = a + b; } \quad \text{sum = a + b; } \quad \text{sum=a+b ;}
\]

are all the same, but \(\text{sum = a+b;}\) is very different (why?).

❖ Use white space wisely, in particular for indentation. Good indentation is as important as good comments!
Example of Indentation

...  
printf("Please enter number of values:\n");
if (scanf("%d", &numbers_to_read) < 1)
{
    printf("Failed reading the number of values.\n");
    return 1;
}
for (i = 0; i < numbers_to_read; i++)
{
    printf("Enter the next integer: ");
    if (scanf("%d", &value) < 1)
    {
        printf("Failed reading value %d.\n", i + 1);
        return 1;
    }
    sum = sum + value;
}
...
Keywords are reserved words that have a **prescribed meaning** and function in C. You **cannot** re-define them.

The ANSI-C standard of 1989 (**C89**) defines only 32 keywords:

- auto
- do
- goto
- signed
- unsigned
- break
- double
- if
- sizeof
- void
- case
- else
- int
- static
- volatile
- char
- enum
- long
- struct
- while
- const
- extern
- register
- switch
- continue
- float
- return
- typedef
- default
- for
- short
- union


In addition to keywords, there are other words that you **can** re-define, but should **never try**, at least not in this course. For example: **main**, **printf**, and **scanf**.
A variable is a “cell” in the main (RAM) memory, consisting of one or more bytes. It usually stores a piece of data needed by the program. The program can read and write its content (value).

An identifier associates a symbolic name with a variable.

- Must consist of just letters [A–Z,a–z], digits [0–9], and _ but cannot begin with a digit.
- Could be as long as you want, but only the first 31 characters matter (C89).
- Identifiers are used to assign symbolic names also to functions, types, etc.

Choose meaningful names as your identifiers!

\[
\text{ECE15\_score} = \text{homework\_average} \times 0.30 + \text{midterm\_grade} \times 0.20 + \text{final\_grade} \times 0.50;
\]
This piece of C code contains **constants** of different types: integer constants, floating-point constants, and character constants.

There are also other types of constants.
Examples of Numerical Constants

❖ The following are *integer constants*:

\[
200 \quad -42 \quad 032 \quad 0x1B7 \quad 72057868917932033UL
\]

Here, \(032\) is an *octal constant* (26 decimal), and \(0x1B7\) is a *hexadecimal constant* (439 decimal). The last constant is too large to fit in the standard *int* data type, so its type *unsigned long* is indicated using the letters *UL*.

❖ The following are *floating-point constants*. They contain either a decimal point or an exponent *e* or both:

\[
2.0 \quad .53 \quad -2.5 \quad 6. \quad 2.0F \quad 2.0L \\
2e-7 \quad -2e+7 \quad -2e7 \quad 3.39e+9 \quad 2e+0F \quad .2e+1L
\]

The letters *F* and *L* override the default type *double* to *float* and to *long double* respectively.

**Observation:** The constants 2 and 2.0 are represented in the main memory (RAM) very differently. Why?
Character and String Constants

❖ **A character constant** is any character enclosed in `'`. For example: 'a', 'Z', '3', '$', etc. Certain useful characters cannot be typed-in, so **escape sequences** are used to represent them:

```
'\n' -- newline   '\t' -- tab   '\b' -- backspace
```

❖ **A string constant** is any sequence of characters enclosed in `""`. For example: "this is a string", "hello world\n", ")%*j!".

- Inside a string constant, everything is interpreted literally simply as a sequence of characters, without any meaning:

```
"a = b + c;"
printf("return 1");
/* this is not a comment */
```

- What is the difference between 'a' and "a" …?
Symbolic Constants and `#define`

- The following line in a program will cause the compiler to replace all occurrences of `CLASS_SIZE` in the program file with `100`:

  ```c
  #define CLASS_SIZE 100
  ```

- With one exception: if `CLASS_SIZE` appears inside of a string constant.
- This is done at the `pre-processing stage`: the very first time the compiler touches the file, before doing anything else.
- Note that `CLASS_SIZE` is not a variable but a constant; no memory is allocated for it, and it cannot change its value at run-time.

**Why define symbolic constants?**

- The class has 100 students today, but may have more or less than that next week. We’ll need to update only one line in the program.
- `CLASS_SIZE` is much more expressive than `100`. Thus the program becomes easier to read and understand.
- Eliminates the need to type-in complicated constants:
Here is a table of all the operator tokens in C, listed in the order of their precedence:

<table>
<thead>
<tr>
<th>Operator Tokens</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>-&gt;</td>
</tr>
<tr>
<td>!</td>
<td>~</td>
</tr>
<tr>
<td>*</td>
<td>/</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;=</td>
</tr>
<tr>
<td>==</td>
<td>!=</td>
</tr>
<tr>
<td>+=</td>
<td>-=</td>
</tr>
</tbody>
</table>

There are many types of operators: arithmetic, logical, relational, bitwise, etc. We will discuss all this and more the following lecture.
Here is a table of all the **punctuator tokens** in C, listed in no particular order:

<table>
<thead>
<tr>
<th>[]</th>
<th>()</th>
<th>{}</th>
<th>,</th>
<th>:</th>
<th>;</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>=</td>
<td>...</td>
<td>#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>-&gt;</td>
<td>++</td>
<td>--</td>
<td>##</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>+</td>
<td>-</td>
<td>~</td>
<td>!</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>%</td>
<td>&lt;&lt;</td>
<td>&gt;&gt;</td>
<td>!=</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>==</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td></td>
<td></td>
<td>&amp;&amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>+=</td>
<td>-=</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>&amp;=</td>
<td>^=</td>
<td></td>
<td>=</td>
</tr>
</tbody>
</table>
A few very useful things:

- Comments and documentation
- Functions and their parameters
- Including header files with `#include`
- Standard library functions `printf()` and `scanf()`

User interface for the simple adder program

Basic building blocks of the C language:

- Keywords and identifiers
- Variables and constants
- Defining symbolic constants with `#define`

Basic Data Types in C: `int`, `char`, `double`
Before a variable can be used in a C program it must be declared. The declaration of a variable contains the following: the name of the variable, its type, and optionally its initial value.

The declared type of a variable determines the following:

- The size of the memory cell (counted in terms of the number of bytes) that the variable will occupy.
- The interpretation/meaning of the binary string stored in that cell.
- Which operations on the variable are permitted and how they work.

The declaration of a variable is an instruction to the compiler to do the following:

- Allocate a memory cell of the appropriate size.
- Interpret the content of this cell according to the given type.
- Associate the variable’s name to this cell.

```c
int num, sum; double weight = 0.0; char digit = '4';
```
The Basic Integer Type

❖ The basic type for representing integers in C is int.
   - For example, all the integer constants are assumed to be of type int, unless specifically indicated otherwise by the programmer.

❖ The ANSI-C (C89) and the ISO-C (C99) standards do not postulate how many bytes should be allocated to type int. However, most compilers today will allocate 4 bytes.

❖ The range of integers that can be represented by the type int is determined by the number of bytes allocated to it:
   - Two bytes (16 bits): \(-2^{15}, \ldots, -2, -1, 0, 1, 2, \ldots, 2^{15} - 1\)
   - Four bytes (32 bits): \(-2^{31}, \ldots, -2, -1, 0, 1, 2, \ldots, 2^{31} - 1\)
   - Eight bytes (64 bits): \(-2^{63}, \ldots, -2, -1, 0, 1, 2, \ldots, 2^{63} - 1\)

Note that \(2^{63} = 9,223,372,036,854,775,808\), \(2^{31} = 2,147,483,648\), and \(2^{15} = 32,768\) (very often not enough).
There are at least three other integer types: `char`, `short`, and `long`. Most compilers also support the type `long long`. However, the only thing that the C standards guarantee is:

\[
1 = \text{char} \leq \text{short} \leq \text{int} \leq \text{long}
\]

Another option is to give up on the representation of negative integers, and thereby increase the available range of positive integers by a factor of 2. This can be accomplished using the keyword `unsigned`. For example, suppose that the basic `int` type occupies 4 bytes. Then

- range of `int` is: \(-2^{31}, \ldots, -2, -1, 0, 1, 2, \ldots, 2^{31}-1\)
- range of `unsigned int` is: \(0, 1, 2, \ldots, 2^{32}-1\)

The types `unsigned char`, `unsigned short`, `unsigned long` work in the same way.
❖ All **characters**, such as 'a', 'Z', '3', '$', are represented internally in C simply as **integers** in the range 0,1,2,...,255. To do this, each character is assigned a numerical code in this range, known as its **ASCII code**. Here is a partial table of ASCII codes:

<table>
<thead>
<tr>
<th>char</th>
<th>letter</th>
<th>=</th>
<th>0</th>
<th>...</th>
<th>48</th>
<th>49</th>
<th>...</th>
<th>65</th>
<th>66</th>
<th>...</th>
<th>97</th>
<th>98</th>
<th>...</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>\0</td>
<td>...</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>A</td>
<td>B</td>
<td>...</td>
<td>a</td>
<td>b</td>
<td>...</td>
<td>DEL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

❖ For example, the following declarations do the same thing, they are completely equivalent:

```
char letter = 'd';
char letter = 100;
```

The only time when such integers are converted to actual characters is during output, if so requested by the programmer:

```
printf("%c", letter);  // d
printf("%d", letter);  // 100
```
#include <stdio.h>

int main()
{
    char value;
    int i;

    for (i = 0; i < 128; i++)
    {
        value = i;
        printf("%d: %c ---> %d\n", i, value, value);
    }
    return 0;
}

**Conclusion:** The `char` data type simply defines an 8-bit integer in the range \(-128,\ldots,-2,-1,0,1,2,\ldots,127\). It is converted to an ASCII character only inside the `printf` function.
The basic type for representing real values in C is `double`.

For example, the real-valued constants are assumed to be of type `double`, unless specifically indicated otherwise by the programmer.

There are at least two other real-valued data types in C, namely `float` and `long double`. The C standards guarantee that:

\[
\text{sizeof(float)} \leq \text{sizeof(double)} \leq \text{sizeof(long double)}
\]

Most compilers allocate 8 bytes to type `double` and 4 bytes to type `float`. Real values are stored in variables of type `float` or `double` using the *floating-point representation*. The size of the data type (number of bytes) determines both the range of the real values that can be represented and the accuracy with which such real values are represented.
Floating-point values are represented as **binary fractions.** Integers (e.g. 42.0) and dyadic fractions (such as 42/1024) are represented exactly. Other real numbers are approximated.

Floating-point values are represented as: $-1^s \times m \times 2^e$

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent</th>
<th>mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11001001000011111</td>
<td>11.00100100001111</td>
</tr>
</tbody>
</table>

3.1415926

Typical range of values for **float** is from $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$ with 7 digits of precision. Typical range of values for **double** is from $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ with 15 digits of precision.